# Динамическая поддержка расширений процессора в кросс-системе

*В.В. Рубанов, А.И. Гриневич, Д.А. Марковцев, М.А. Миткевич*

**Аннотация**. В статье рассматривается задача моделирования расширений процессора в виде акселераторов (сопроцессоров) для автоматической настройки кросс-системы, включающей в себя ассемблер, компоновщик, симулятор, отладчик и профилировщик. Рассматриваемый подход основан на описании моделей акселераторов на разработанном языке спецификации с последующим использованием этих моделей кросс-системой во время ассемблирования, потактовой симуляции и отладки прикладных программ, содержащих команды акселераторов, не известные на этапе построения основного инструментария (определяемые соответствующими моделями). В статье рассматривается математическая модель поддерживаемых акселераторов и интерфейса с основным процессором. Описывается язык спецификации конкретных моделей акселераторов, дается обзор технологий настройки кросс-системы для поддержки описанных таким образом расширений.

## *1. Введение*

В статье рассматривается задача моделирования расширений процессора в виде *акселераторов* (сопроцессоров) для автоматической настройки *инструмен-тария кросс-разработки* для поддержки этих расширений. Под инструмен-тарием кросс-разработки (кросс-системой) понимается набор программных компонентов (ассемблер, компоновщик, симулятор, отладчик и профилиро-вщик) для разработки прикладных программ с использованием хост-машины, отличной от целевой аппаратуры. Под поддержкой расширений кросс-системой подразумевается ассемблирование, потактовая симуляция и отладка прик-ладных программ, содержащих команды, не известные на этапе построения основного инструментария (реализуемых специфическими для пользователя акселераторами). Рассматриваемый подход основан на предоставлении пользо-вателю возможности описать модели акселераторов на разработанном языке спецификации с последующим использованием этих моделей для настройки компонентов кросс-системы.

Данная задача возникает в связи с тем, что многие современные аппаратные решения строятся на основе использования стандартного процессорного ядра со специализированными расширениями в виде акселераторов. Часть системы команд ядра зарезервирована для команд обращения к интерфейсу запуска инструкций акселераторов. Однако семантика реальных действий и вычис-лений, которые инициируют такие команды, определяется



конкретными акселераторами и не зависит от основного процессора. Производитель ядра и производители акселераторов могут быть разными компаниями, при этом инструментарий кросс-разработки от производителя основного процессора должен уметь поддерживать неизвестные для него расширения аппаратуры, которые создаются заказчиками. В данной работе под расширениями процессора понимается добавление акселераторов, которые могут вводить в систему новые элементы памяти (регистры, памяти данных) и определять семантику команд запуска инструкций акселераторов. Аппаратура на базе основного процессора с акселераторами представляется для прикладного программиста как вычислительная система с единой системой команд и одной программой. Память системы состоит из памяти основного процессора, разделяемой памяти и локальных памятей акселераторов.

Для выделения класса поддерживаемых акселераторов и интерфейса с процессором была разработана абстрактная математическая модель, позво-ляющая моделировать состояние и поведение широкого класса акселераторов с потактовой точностью. Для описания конкретных моделей, в рамках данной абстрактной, предложен язык спецификации (ISE), разработаны средства визу-ального редактирования спецификаций на этом языке и средства анализа и выявления ошибок в спецификациях. Реализован подход интерактивной пере-настройки кросс-системы, заключающийся в настройке компонентов (ассемблер, дисассемблер, симулятор, отладчик) на основании интерпретации описаний моделей акселераторов. Система используется в коммерческой эксплуатации, в ней успешно реализованы модели реальных акселераторов.

Статья состоит из введения, трех глав и заключения. В первой главе описы-вается подход к моделированию акселераторов, вводится абстрактная модель акселератора и средства описания конкретных моделей на языке ISE. Во второй главе содержится описание технологии интерактивной перенастройки кросс-системы для поддержки заданных пользователем моделей акселератов. Третья глава содержит краткий обзор смежных работ. В заключении приводятся практические результаты, полученные при эксплуатации разработанной системы. Приводится план будущих направлений развития темы.

## *2. Моделирование расширений процессора*

В данной главе рассматриваются вопросы моделирования расширений процес-сора в виде акселераторов. Для этого вводится абстрактная (математическая) модель акселератора, охватывающая достаточно широкий класс возможной аппаратуры (см. 2.1). В рамках этой абстрактной модели определяются сред-ства описания конкретных моделей акселераторов в виде файлов на специа-льном языке ISE (см. 2.2). Описанные таким образом модели акселераторов используются для автоматической конфигурации компонентов кросс-системы – симулятора, ассемблера, дисассемблера, отладчика и профилировщика (см. главу 3).

## 2.1. Абстрактная модель акселератора

В нашей модели акселератор является сопроцессором, команды которому выдаются основным процессором. Акселератор работает синхронно с процессором. Акселератор может иметь собственную локальную память и имеет доступ к общей памяти между процессором и всеми акселераторами.

### *2.1.1. Состояние акселератора*

#### 2.1.1.1. Память акселератора

*Ячейка памяти* представляет собой набор двоичных переменных (далее *битов*), с возможными значениями 0 или 1. Число битов определяет *разрядность* ячейки. Набор из одной или более ячеек одинаковой разрядности образует *область памяти*. Набор из одной или более областей образует *память*. Память будем обозначать большой буквой S. *Состояние* ячейки памяти определяется набором конкретных значений всех ее битов. *Состояние памяти* определяется состоянием всех ячеек ее областей. Состояние памяти будем обозначать маленькой буквой s. Обозначим через N сумму разрядностей всех ячеек памяти S, тогда память может находиться в одном из $2^N$ состояний. Множество состояний памяти будем обозначать {s}. Заметим, что это множество однозначно задается *структурой памяти*. *Дескриптор структуры памяти* $\Sigma$ представляет собой следующий набор чисел: число областей O и набор из O пар ($W_i$, $S_i$), задающих разрядность $W_i$ ячеек области и их количество $S_i$.

В нашей модели *память системы* состоит из *памяти основного процессора* $S^P$, *разделяемой памяти* $S^S$ и *локальных памятей акселераторов* $S^a$:

$$S = \left\{ S^P, S^S, \{S_i^a\}_{i=1}^{N^A} \right\},\ N^A \text{ - число акселераторов}$$

Рассмотрение памяти процессора не принципиально для данной статьи, так как акселератор имеет доступ только к разделяемой и своей локальной памяти. Пара из этих памятей формирует полную *память акселератора*:

$$S^A = \{S^S, S^a\}$$

Множество возможных состояний памяти акселератора обозначим как $\{s\}_A = \{s\}_S \times \{s\}_a$. Каждая область в локальной и разделяемой памяти характе-ризуется *скоростью доступа* - числом, означающим, сколько тактов проходит после записи в ячейку этой области, прежде чем измененное значение может быть прочитано; до этого момента при чтении считывается старое значение.

Обычно в памяти акселератора можно выделить следующие области:

- Набор из одной или более *памятей данных*
- Набор из одного или более *регистровых файлов*
- *Одиночные регистры*



#### 2.1.1.2. Управляющее состояние

Акселератор имеет фиксированное количество *управляющих слотов*, совокупность которых обозначается $P^A$. Каждый слот имеет номер, который далее будет отождествляться с соответствующим слотом. Каждый слот $L_i$ представляет собой пару целочисленных переменных (полей): поле *номера команды* $n_f$ и поле *состояния команды t*. Множество значений поля команды $\{n_f\}$ конечно. Ноль всегда принадлежит $\{n_f\}$. Существует взаимно однозначное соответствие между $\{n_f\}\backslash 0$ и множеством дескрипторов команд, определение которого будет дано ниже. Поэтому мы будем отождествлять номер $n \in \{n_f\}$ и соответсвующий дескриптор *f*. Множество значений поля состояния команды есть натуральные числа (включая 0). Состояние слота с номером n обозначается $l_n = (n_f, t)$ и определяется значениями полей этого слота. Множество возможных состояний слота $\{l\} = \{n_f\} \times \mathrm{N}$. Слот в состоянии (0, 0) называется *свободным*, в ином случае *активным*. *Управляющим состоянием p* акселератора называется совокупность состояний всех его управляющих слотов. *Множество управляющих состояний* акселератора обозначим $\{p\}_A = \{l\}^{N_S}$, где $N_s$ число слотов акселератора. Активные слоты соответствуют выполняющимся командам акселератора, число управляющих слотов задает максимальное количество параллельно выполняющихся команд. В состоянии активного слота значение поля номера команды задает выполняющуюся команду, а значение поля состояния команды соответствует количеству тактов, прошедших с начала выполнения этой команды.

#### 2.1.1.3. Состояние акселератора

***Состояние акселератора*** *a* задается парой из состояния памяти и управляющего состояния: a={s, p}. *Множество состояний акселератора* обозначим как $\{a\}_A = \{s\}_A \times \{p\}_A$.

*Элементом состояния* акселератора называется любая ячейка памяти акселератора или любой управляющий слот. Состоянием элемента называется соответственно состояние ячейки или слота. *Параллельной композицией* функций $\varphi_1, \varphi_2 ... \varphi_n$ (заданных на множестве состояний акселератора) назовем функцию $\omega = \varphi_1 \parallel \varphi_2 \parallel ... \parallel \varphi_n$ (также заданную на множестве состояний акселератора), получаемую следующим образом: пусть $\mathrm{A} = \{\alpha\} = S^A \cup P^A$ множество всех элементов состояния акселератора, $\mathrm{A}_i$ - множество элементов, состояние $a_\alpha$ которых было изменено функцией $\varphi_i$: $\alpha \in \mathrm{A}_i \Leftrightarrow a_\alpha \neq \varphi_i(a_\alpha)$.

Если пересечение всех $A_i, i = 1..n$ не пустое множество, то значение функции $\omega$ не определено. В ином случае значение $\omega$ задается следующим образом:

$$a' = \omega = \begin{cases} a'_\alpha = a_\alpha, \alpha \in A \setminus Y A_i \\ a'_\alpha = \varphi_i(a), \alpha \in A_i \end{cases}$$

, здесь $a'_\alpha$ и $a_\alpha$ - новое и старое состояние элемента $\alpha$ соответственно

### *2.1.2. Модель акселератора*

Операцией [элементарной] называется, заданная на множестве состояний памяти акселератора $\{s\}_A$ (см. 2.1.1.1) функция $\psi$, формирующая следующее состояние памяти акселератора на основе предыдущего. Операции соответствуют действиям, которые могут быть выполнены за один такт (например, сложение двух регистров):

$$\psi : \{s\}_A \to \{s\}_A$$

Множество операций акселератора обозначим $\Psi^A = \{\psi\}$. Для любого акселератора в этом множестве присутствует так называемая *пустая операция*, не изменяющая состояние и обозначаемая $\psi_0$.

Каждая элементарная операция характеризуется *функциональными ресурсами*, которые необходимы для выполнения этой операции. Обозначим множество всех ресурсов акселератора как $R^A$, а множество всех его подмножеств (включая пустое) как $P^A$. Функция $r^A$, отображающая множество операций $\Psi^A$ на множество $P^A$, называется *функцией ресурсов*. Она задает набор ресурсов для каждой операции в $\Psi^A$:

$$r^A : \Psi^A \to P^A$$

Обозначим множество всех подмножеств в $\Psi^A$, которые состоят из операций, использующих не пересекающиеся ресурсы, как $\Omega^A$:

$$\omega \in \Omega^A \Leftrightarrow \forall \psi_i, \psi_j \in \omega, i \neq j \to r^A(\psi_i) \cap r^A(\psi_j) = \varnothing$$

Таким образом, элемент множества $\Omega^A$ задает группу операций $\{\psi_i\}$, которые могут выполняться параллельно (в рамках одного такта). Применением параллельной композиции (см. выше) этот элемент задает функцию на множестве состояний памяти акселератора $\{s\}_A$. Далее элементы множества $\Omega^A$ будем отождествлять с задаваемыми ими функциями и называть ***комплекс-операциями***. Заметим, что в этом смысле $\Psi^A \subset \Omega^A$.

Определим два ***управляющих действия***: продвижения *next* и окончания *end*. Каждое представляет собой параметрическую функцию на множестве



управляющих состояний акселератора. Параметром функции является номер слота $n_l \in [1..N_S]$. Действие *next* увеличивает на 1 значение поля состояния команды *t* для слота, заданного параметром $n_l$.

$$p' = next(n_l, p) = \begin{cases} l'_n = l_n, n \neq n_l \\ l'_n = (f_n, t_n + 1), n = n_l \end{cases}$$

Действие *end* переводит соответствующий слот в холостое состояние.

$$p' = end(n_l, p) = \begin{cases} l'_n = l_n, n \neq n_l \\ l'_n = (0, 0), n = n_l \end{cases}$$

Множество из этих управляющих действий обозначим $U^A = \{next, end\}$.

***Дескриптором команды акселератора*** называется функция $f$, вычисляющая пару из комплекс-операции $\omega \in \Omega^A$ и управляющего действия $u \in U^A$ на основании состояния памяти акселератора $s \in \{s\}_A$ и состояния команды $t \in N$ (см. 2.1.1.2):

$$f : \{s\}_A \times N \to \Omega^A \times U^A$$

Множество дескрипторов команд акселератора обозначается $I^A = \{f_i\}$. Это множество конечно и каждому элементу $f$ приписывается номер $n_f \geq 1$ (например, в порядке возрастания машинного кода в соответствии с отображением, задаваемым функцией декодирования, см. ниже). При этом для дескриптора команды всегда верно следующее (единственность конца команды):

$$f(s, \tau) = \{\omega, end\} \Rightarrow \forall t > \tau : f(\omega(s), t) = undefined$$

Дескриптор команды $f$ однозначно задает *функцию потактового поведения команды* $b_f$, определенную на множестве состояний акселератора и параметризуемую номером слота $n_l$. Значение функции $b_f$ не определено, если значение поля номера команды в соответствующем слоте $n_l$ не совпадает с номером соответствующего дескриптора $f$. В ином случае функция $b_f(n_l) : \{a\}_A \to \{a\}_A$ задается следующим образом:

1. На основании состояния памяти s и значения поля состояния команды t в слоте с номером $n_l$ с помощью дескриптора команды определяется пара из комплекс-операции $\omega$ и управляющего действия $u$:
$$\{\omega, u\} = f(s, t)$$

2. С помощью $\omega$ определяется следующее состояние памяти $s' = \omega(s)$, а управляющее действие задает следующее управляющее состояние $p' = u(n_l, p)$. Пара $(s', p')$ задает значение $b_f(n_l, s, p)$.

Таким образом, множество дескрипторов команд однозначно задает множество функций потактового поведения $B^A$, имеющее столько же элементов.

***Множество кодов инструкций акселератора*** $C^A = \{c_i\}$ представляет собой множество двоичных чисел одинаковой разрядности. Каждый элемент этого множества соответствует одному из возможных значений поля *код инструкции акселератора* в машинном коде команды запуска акселератора (см. 2.1.3.1).

***Функция декодирования*** $d^A$ задает отображение множества кодов инструкций акселератора $C^A$ на множество дескрипторов команд $I^A$:

$$d^A : C^A \to I^A$$

Функция декодирования однозначно задает *декодер акселератора* – функцию $D^A$:

$$D^A : C^A \times \{p\}_A \to \{p\}_A :$$

Функция $D^A$ определяется следующим образом: если $\exists n_0 : l_{n_0} = (0, 0)$, то $p' = D^A(c, p) = \begin{cases} l'_n = (f_n, t_n), & n \neq n_0 \\ l'_n = (d^A(c), 1), & n = n_0 \end{cases}$, иначе значение $D^A$ не определено.

Иными словами, в случае наличия свободного управляющего слота декодер инициирует состояние этого слота в $(f, 1)$, где $f$ определяется по коду инструкции с помощью функции декодирования $d^A$.

***Тактовая функция*** $T^A$ на множестве состояний акселератора $\{a\}_A$, определяет изменение состояния акселератора на каждом такте:

$$T^A : \{a\}_A \to \{a\}_A$$

Эта функция однозначно задается параллельной композицией (см. 2.1.1.3) функций потактового поведения $b_f$, соответствующих дескрипторам команд $f$, заданным в поле $n_f$ каждого *активного* управляющего слота. Иными словами для каждого такта управляющее состояние акселератора определяет набор активных команд, характеризуемых функциями потактового поведения. Параллельная композиция этих функций задает отображение $T^A$ (поведение акселератора) для текущего такта. Если активных слотов нет, то состояние акселератора не меняется.



***Абстрактная модель акселератора*** $M^A$ определяется следующими описанными выше элементами:

$$M^A = \left\{S^A, P^A, C^A, D^A, T^A\right\}$$

### *2.1.3. Симуляция акселератора*

Для *симуляции* акселератора, заданного моделью $M^A$ (см. 2.1.2), необходим *генератор тактов*, а также определенное *начальное состояние памяти* $s_0 \in \{s\}_A$. В начальном управляющем состоянии $p_0$ все слоты свободны. В рассматриваемой системе акселераторы и основной процессор работают тактово-синхронно (тактовый генератор единый для всей системы), то есть такт работы акселератора равен такту работы процессора. Кроме тактового генератора, единственным внешним событием для акселератора является выдача очередной команды основным процессором (см. 2.1.3.1 ниже).

#### 2.1.3.1. Запуск команд акселератора

В рассматриваемой модели аппаратуры множество команд основного процессора должно иметь непустое подмножество, представляющее собой *команды запуска акселераторов*. Такие команды инициируют запуск определенной инструкции соответствующего акселератора. С точки зрения основного процессора команда запуска акселераторов определяется тремя полями машинного кода (порядок полей несущественен, также поля не обязательно должны быть непрерывными):

`{КОП, номер акселератора, код инструкции акселератора}`

Действия основного процессора при выполнении команды запуска акселератора заключаются в активации акселератора с номером в соответствующем поле и выдаче этому акселератору *кода инструкции акселератора* для дальнейшего декодирования и выполнения команды в самом акселераторе параллельно с работой процессора. Для основного процессора выполнение команды запуска акселератора всегда занимает один такт. В терминах абстрактной модели выдача команды акселератора основным процес-сором заключается в передаче кода инструкции акселератора в функцию деко-дера акселератора $D^A$ ($C^A$ это подмножество множества значений поля `код инструкции акселератора`). За один такт процессор может выдать не более одной команды акселератора. Заметим, что код инструкции акселератора в свою очередь может содержать КОП команды акселератора и операнды.

Акселератор может параллельно выполнять несколько многотактовых команд, в том числе с одинаковым дескриптором. То есть основной процессор может выдавать новую команду акселератора, до того как отработали предыдущие команды. В рамках рассматриваемой модели это возможно, если все операции, выполняемые параллельно на каждом такте, используют непересекающиеся ресурсы (см. 2.1.2). На практике это

возможно, если позволяет конвейер и функциональные устройства акселератора, при этом ответственность за корректную (своевременную) выдачу команд акселератора лежит на прикладном программисте.

#### 2.1.3.2. Тактовое поведение акселератора

В ответ на событие от тактового генератора, в рассматриваемой модели действия акселератора сводятся к изменению состояния в соответствии со своей тактовой функцией $T^A$. Эта функция определяет поведение акселератора на каждом такте.

#### 2.1.3.3. Обмен данными и синхронизация с процессором

Обмен данными между процессором и акселераторами осуществляется через разделяемую (общую) память (см. 2.1.1.1). Заметим, что дополнительная информация от процессора к акселератору может также поступать в виде параметров инструкции (см. 2.1.3.1). Заметим, что разные акселераторы не имеют доступа к локальной памяти друг друга.

Разделение доступа к общей памяти в нашей модели соответствует типу CREW (Common Read Exclusive Write). Это означает, что процессор и акселераторы могут одновременно (в рамках текущего такта) читать из ячейки памяти, однако одновременная запись запрещена. В рассматриваемой модели области памяти могут иметь задержку записи, характеризуемую скоростью доступа к памяти (см. 2.1.1.1). По умолчанию, все области памяти имеют задержку 1, то есть изменения могут быть прочитаны только на следующем такте (flip-flop модель). Заметим, что если задержка больше ноля, то возможна одновременная запись и чтение одной и той же ячейки, при чтении считывается предыдущее значение.

Команды акселератора могут занимать фиксированное или переменное (в зависимости от данных) число тактов. С точки зрения прикладного программиста (компилятора) существует три способа синхронизации вычислений основного процессора и результатов работы определенной команды акселератора:

1. Когда команда акселератора всегда имеет фиксированное число тактов выполнения, программист может статически просчитать, когда будут готовы результаты вычислений (процессор и акселераторы работают синхронно, см. 2.1.3).

2. Акселератор в процессе выполнения может выставлять определенные флаги (менять ячейки) в общей памяти. Программа основного процессора может считывать значения этих флагов и определять готовность результатов вычислений акселератора.

3. Частным случаем пункта 2 является вызов акселератором прерывания основного процессора. Обработчик прерывания может прочитать результаты вычислений акселератора.



## 2.2. Средства описания конкретных моделей акселераторов

Для задания ***конкретной модели акселератора*** необходимо определить следующие параметры, множества и функции:

$$m^A = \left\{ \Sigma^A, N^S, C^A, d^A, I^A, \Psi^A, R^A, r^A \right\}$$

Соответствующие определения были даны в 2.1.1 и 2.1.2, где также было показано, что они однозначно задают все элементы абстрактной модели: $\Sigma^A \to S^A$, $N^S \to P^A$, $d^A \to D^A$, $\Psi^A, R^A, r^A \to \Omega^A$, $I^A, \Omega^A \to B^A \to T^A$.

Для описания конкретных моделей акселераторов в ИСП РАН был разработан язык спецификации ISE (Instruction Set Extension). Кроме собственно спецификации соответствующих элементов конкретной модели акселератора (см. выше), в язык также входят средства описания дополнительной информации об ассемблерном синтаксисе команд акселератора, отображении ассемблерных команд в машинные коды и описание форматов для визуализации областей памяти в отладчике. Модель акселератора далее будет отождествляться со спецификацией этой модели на языке ISE.

### *2.2.1. Дескриптор структуры памяти*

Существуют отдельные синтаксические конструкции для описания следующих типов областей памяти:

- Памяти данных
- Регистровые файлы
- Одиночные регистры

С помощью этих конструкций можно задать дескриптор структуры памяти $\Sigma^A$ (см. 2.1.1.1).

**Пример**. Структура памяти простого акселератора:

- две памяти данных LDM и TM размером 2048 слов каждая с разрядностью 16 и 64 бита со скоростью доступа 3 такта
- Регистровый файл GRF из двух 16-ти разрядных регистров GR0 и GR1
- Одиночный регистр-аккумулятор ACR разрядностью 36 бит

```
    DECLARE_MEMORY(INT(16, 3), 2048) LDM;
    DECLARE_MEMORY(INT(64, 3), 2048) TM;
    DECLARE_REGISTERS_FILE(INT(16), 2) GRF;
    DECLARE_REGISTER(UINT(36)) ACR;
    // debugging names and registers file structure
    MEMORY(LDM, "Acc LDM");
    MEMORY(TM, "Acc TM");
REGFILE_BEGIN(GRF, "General Registers")
```

```
    REGISTER(0, "GR0");
    REGISTER(1, "GR1");
REGFILE_END()
```

### *2.2.2. Описание поведения*

К семантике поведения акселератора относятся следующие элементы модели $m^A$: множество ресурсов $R^A$, функция ресурсов $r^A$, множество операций $\Psi^A$, множество дескрипторов команд $I^A$ и функция декодирования $d^A$ (вместе с $C^A$).

#### 2.2.2.1. Операции

Для задания операций из $\Psi^A$ используется язык C++. Ячейки памяти акселератора доступны в качестве глобальных переменных (регистровые файлы и памяти в виде массивов). Для удобства описания могут объявляться собственные локальные переменные. Также могут быть использованы возможности специальной библиотеки (например, N-битные типы данных `INT<N>`, `UINT<N>`, типы данных с фиксированной точкой `FIXED<I,F>`, операции битовых манипуляций и т.п.). Используемые в операции ресурсы обозначаются в виде вызова функции `UseResources(resources)` (тем самым задается функция $r^A$). В существующей реализации список используемых в данной операции ресурсов передаются в виде битовой строки, где каждый ресурс соответствует определенному биту. Множество ресурсов $R^A$ задается в виде перечисления (`enum`) со значениями элементов по степеням двойки:

```
enum Resources {MAC_ADDER=1, MAC_MULTIPLIER=2,
ALU_ADDER=4};
```

**Пример 1**. Операция по сложению двух 36-ти разрядных чисел:

```
void ADD_36_36(INT<36>& res, INT<36> a, INT<36> b) {
 UseResources(MAC_ADDER);
 res = a + b;
}
```

**Пример 2**. Операция по перемножению двух 16-ти разрядных знаковых чисел:

```
void SMUL_16_16(INT<32>& res, INT<16> a, INT<16> b)
{
 UseResources();
 res = a * b;
}
```



Заданная на C++ операция может быть оформлена в виде отдельной функции (см. примеры выше) или встраиваться непосредственно в тело функции поведения команды (см. примеры в 2.2.2.2).

#### 2.2.2.2. Дескрипторы команд

Дескрипторы команд акселератора из $I^A$ задаются соответствующими *функциями поведения команд*. Функция поведения может принимать аргументы в виде параметров инструкции $p_i$. Тем самым одна функция поведения может описывать набор дескрипторов (один дескриптор соответствует одному конкретному набору значений параметров). Тело функции поведения может описываться на языке C++. Отображение в операции для соответствующих значений состояния команды t неявно задается путем использования специальной функции `FinishCycle()`. Вызовы данной функции отделяют операции внутри функции поведения, относящиеся к последовательным тактам исполнения (значениям параметра t дескриптора команды). Для описания динамического характера выбора операций в зависимости от состояния акселератора (аргумент дескриптора s) в описании функции поведения команды допускается использование управляющих конструкций языка C, в частности циклов и ветвлений (см. пример 3 ниже). Вызов функции `FinishCycle()` означает окончание всех операций для текущего такта команды и соответствует управляющему действию `next`. Возврат из функции поведения команды соответствует управляющему действию `end`. Использо-вание такого решения позволяет эффективно описывать дескрипторы команд, тем самым определяя потактовое поведение команд акселератора.

**Пример 1**. Однотактовая команда перемещения между регистрами, содержащая единственную операцию, задаваемую конструкцией `GRF[greg] = LRF[lreg]`:

```
ACC_FUNCTION Move_LREG_GREG(INT<4> lreg, INT<4>
greg) {
 GRF[greg] = LRF[lreg];
 FinishCycle();
}
```

**Пример 2**. Двухтактовая команда перемножения и аккумуляции результата. На первом такте происходит перемножение операндов (операция SMUL_16_16 – см. пример 2 в 2.2.2.1), на втором – аккумуляция результата (операция ADD_36_36 – см. пример 1 в 2.2.2.1):

```
ACC_FUNCTION MAC_LREG_GREG(INT<4> grs, INT<4> grt) {
 SMUL_16_16 (mulres, GRF[grs], GRF[grt]);
 FinishCycle();
 ADD_36_36 (ACC, ACC, mulres);
 FinishCycle();
}
```

Заметим, что две выдачи подряд этой команды процессором приведут к ситуации, когда одновременно будут исполняться две различные стадии этой функции (стадия умножения второй команды и стадия сложения первой команды). Такой эффект может быть использован для моделирования конвейера акселератора.

**Пример 3**. Команда свертки векторов, расположенных в памятях DM0 и TM0. Длительность команды зависит от данных (длина векторов задается регистром LOOPREG). Заметим, что в теле цикла за один такт выполняются несколько операций, использующих непересекающиеся ресурсы. Для синхронизации с процессором используется механизм прерывания:

```
ACC_FUNCTION CONV_ACC_DM0_TM0(INT<4> dreg, INT<4> treg)
  {
    SMUL_16_16 (mulres, DM0[AR[dreg]++],
  TM0[AR[treg]++]);
    FinishCycle();
    while (LOOPREG>0) {
    ADD_36_36 (ACC, ACC, mulres);
        SMUL_16_16 (mulres,
         DM0[AR[dreg]++], TM0[AR[dreg]++]);
    LOOPREG--;
    FinishCycle();
          }
    ADD_36_36 (ACC, ACC, mulres);
        InterruptProcessor();
         FinishCycle();
    }
```

#### 2.2.2.3. Функция декодирования

Функция декодирования $d^A$ задается описанием множества пар из *формата машинного слова* команды и ссылки на функцию поведения команды:

```
INSTRUCTION(<format_string>, <invoker_name>);
```

Формат машинного слова команды задается строкой в следующем алфавите:

1. Битовые символы: ‘0’ и ‘1’
2. Параметрические символы: ‘A-Z’ и ‘a-z’
3. Групповой символ: ‘*’
4. Разделительный символ: ‘-‘

Символы из пунктов 1-3 называются *значимыми символами*. Заметим, что число значимых символов в строке формата команды должно быть равно разрядности машинного слова в системе.

Непрерывная цепочка параметрических символов задает *операнд*. Декодер акселератора выделит указанные биты и передаст полученное значение в функцию поведения команды в виде параметра $p_i$. Различные операнды



разделяются групповым или разделительным символом. Операнды нумеруются в порядке справа налево.

Битовые символы задают фиксированные значения в соответствующих позициях машинного слова. На месте параметрических и групповых символов в машинном коде команды может быть любое битовое значение. Разделительные символы используются для косметических целей, а также для отделения подряд идущих операндов.

**Пример**:

```
INSTRUCTION("11-**-0000-0000-0001-LREG-GREG",
Move_LREG_GREG);
```

Функция поведения MoveLREG_GREG (см. пример 1 в 2.2.2.1) имеет два параметра по 4 бита каждый (LREG[4;7] и GREG[0;3]). Биты [20;21] могут принимать любые значения для данной команды (в данном примере эти биты относятся к коду акселератора и используются командой запуска акселератора основного процессора). Остальные биты фиксированы и составляют КОП инструкции аскелератора.

Заметим, что совокупность всех строк форматов машинного слова задает множество допустимых кодов инструкций данного акселератора ( $C^A$ в 2.1.2).

Число управляющих слотов $N^S$ задается директивой SLOTS(<$N^S$>).

## 2.3. Ассемблерный синтаксис команд акселератора

Отдельная секция файла описания модели акселератора отвечает за определение синтаксиса ассемблера для системы команд этого акселератора. Более точно, эта секция определяет синтаксис для подмножества системы команд основного процессора, соответствующего командам запуска инструкций этого акселератора (см. 2.1.3.1). Это полезно для адекватного отра-жения на уровне ассемблера соответствующей семантики команд, так как одни и те же (в смысле машинных кодов) команды основного процессора (для запуска акселераторов) могут иметь разную семантику, в зависимости от конкретной конфигурации акселераторов в системе, специфичной для каждого заказчика. В дополнение к определению семантики команд акселераторов (см. 2.2), возможность настраивать синтаксис для этих команд является важной функциональностью рассматриваемой системы, позволяющей легко получать готовую для производственного использования кросс-систему, адаптированную для специфичной конфигурации «процессор + акселераторы пользователя».

Описание ассемблерного синтаксиса системы команд состоит из трех секций:

1. Секция типов операндов и псевдонимов
2. Секция команд
3. Секция ограничений (constraints)

Детальное рассмотрение средств описания синтаксиса команд выходит за рамки данной статьи. Ниже приводится только краткий обзор основных возможностей.

### *2.3.1. Отображение ассемблерных команд в машинное слово*

Общий шаблон допустимого ассемблерного синтаксиса для команд акселератора задается в виде:

```
command      ::= mnemonic [parameter {,
parameter}*]
         {|| mnemonic [parameter {, parameter}*]}*

mnemonic     ::= const_string

parameter    ::= operand
      {[const_string] [operand]}*

operand      ::= const_string

const_string ::= любой текст без запятых и
пробелов
```

Примеры возможных команд:

```
DMOVE ACR1.h, DM0(DA0--), ACR1.L, TM0(TA0++)
MOVE GRA, DM1(TA0+25) || ADD GR3, ACR2.H
```

На языке ассемблера команда состоит из мнемоники (нескольких мнемоник для параллельных команд) и набора параметров, разделенных запятой. Каждый параметр может содержать несколько частей - операндов, принадлежащих к каким-либо из описанных типов. В рамках одного параметра операнды должны отделяться непустыми строками константных символов. Комбинация мнемоник ассемблерного синтаксиса отображается в поле КОП соответс-твующей команды. Операнды (operand) отображаются на поля-операнды машинного слова. Возможно задание отображения на не непрерывные поля (когда биты поля перемежаются битами других полей).

**Пример:**

```
 .types
grn    [gr0:0] [gr1:1] [gr2:2] [gr3:3]
const6b $ -32 31

 .mnemonics
 MOVE {grn#0;2},{const6b#2;4#8;2} % 0xA8C0 0xFCC0
```

Описание задает команду MOVE с двумя операндами. Первый операнд типа grn является регистром общего назначения, код регистра размещается в 2х битах начиная с 0-го. Второй операнд является константой в диапазоне [-32;



31] и располагается в машинном слове в двух частях: в 4х битах, начиная со 2-го, и 2х битах, начиная с 8-го. КОП равен 1010-10XX-11XX-XXXX.

### *2.3.2. Ограничения*

#### 2.3.2.1. На отдельные операнды

Ограничения на отдельные операнды определяются соответствующим типом, к которому принадлежит операнд. Однако для наглядной генерации ошибок можно задать более широкий тип для операнда и воспользоваться механизмом общих ограничений (см. 2.3.2.2) для выделения допустимых значений.

#### 2.3.2.2. На взаимосвязь операндов команды

Ограничения на взаимосвязь операндов команды задаются в виде набора логических высказываний относительно сравнений арифметических выражений, которые могут содержать значения любых использованных в команде операндов. Считается, что ограничения выполнены, если все логические высказывания для данной инструкции принимают значение «истина».

Выражения ограничений поддерживают следующие операции:

| | |
|---|---|
| Логические операции | &&, \|\|, ! |
| Операции сравнения | <, >, <=, >=, ==, <> |
| Арифметические операции | +, -, *, /, % |
| Битовые операции | \|, &, ^, ~ |

В выражениях могут быть использованы числовые константы и, собственно, ссылки на операнды. Если несколько операндов имеют одинаковый тип, то следует использовать псевдонимы типов для различения операндов в выражениях ограничений.

**Пример.** Операнды 1 и 2 не должны быть равны:

```
ADD {GRs#8;4}, {GRt#4;4} % 0x796000  0xf9f000
(GRs <> GRt) % "Operands must be different for ADD"
```

Заметим, что с помощью описанного механизма также можно задавать ограничения на отдельные операнды. Таким образом, возможны две стратегии работы с ограничениями на значения операндов: с одной стороны можно создать систему общих типов и затем сужать множества значений с помощью логических высказываний, а с другой – работать с большим количеством специфических типов. В первом случае – можно достичь более конкретных сообщений об ошибках в операндах инструкции, во втором – упрощается описание системы команд. Пользователь может выбрать любую стратегию.

#### 2.3.2.3. На комбинации команд

Каждой команде можно назначить некоторый *набор свойств* и задать для них значения и *области активации*. В качестве значения свойства может выступать либо константа, либо значение одного из операндов команды. Область активации задает диапазон соседних команд, на котором данное свойство активно. Область активации по умолчанию [1;1] затрагивает только текущую команду. Механизм описания свойств (дополненный *предикатами совместимости* – см. ниже) фактически является модифицированным описанием таблиц использования ресурсов (reservation tables).

**Пример:**

```
MAC {acr#14;1},{grs#4;4},{grt#0;4} % 0x6c2000
0xffb800
  [read_grn:grs, read_grn:grt, write_acr:acr:2;2]
```

Данная команда обладает следующими свойствами:

- read_grn – двойное свойство со значениями равными значениям операндов grs и grt. Область активации по умолчанию затрагивает только текущую команду (здесь означает, что значения регистров, заданных операндами grs и grt, читаются на первом такте).
- write_acr – значение свойства равно значению операнда acr. Область активации [2;2] затрагивает следующую команду (здесь означает, что значение acr будет записано на втором такте).

На механизме описания свойств базируется способ задания ограничений на использование ресурсов. Задается список *предикатов совместимости свойств*. Предикат совместимости свойств задает в квадратных скобках набор из пар свойств (пары разделяются запятыми, свойства в паре знаком =). Предикат истинен для пары команд, когда выполняются следующие условия: первая команда обладает всеми свойствами из левых частей пар, вторая обладает всеми свойствами из правых частей пар; при этом для каждой пары значения свойств совпадают на пересечении их областей активации. Предикаты совместимости оцениваются ассемблером для всех пар команд при ассемблировании программы, таким образом обнаруживаются конфликтующие команды. Заметим, что оценки работают гарантированно корректно только на линейных участках.

**Пример:**

```
[write_acr=read_acr] % warning: "WAR conflict for
ACRs"
```

Данный предикат будет верен, если у пары команд значение свойства `write_acr` первой команды совпадет со значением свойства `read_acr` второй команды на пересечении областей активации этих свойств. В данном примере это отражает конфликт по данным (по аккумуляторным регистрам) типа WRITE AFTER READ.



Зарезервировано специальное свойство «any», которым по умолчанию обладает любая инструкция. [any=X] дает истинный предикат, если вторая инструкция обладает свойством X (независимо от его значения).

### *2.3.3. Сообщения об ошибках*

#### 2.3.3.1. Ошибки симуляции

Число $N^S$ (см. 2.1.1.2) определяет максимальное количество параллельных команд в акселераторе. Симулятор генерирует ошибку, если процессор пытается выдать команду, когда нет свободных слотов.

Другой механизм обнаружения ошибок выполнения основывается на использовании *ресурсов* (см. 2.1.2). Если в пределах одного такта один и тот же ресурс используется разными параллельно выполняющимися командами, симулятор генерирует ошибку выполнения.

Для обеспечения CREW модели доступа к данным симулятор обнаруживает запись разными процессами в одну и ту же ячейку памяти и генерирует ошибку выполнения.

#### 2.3.3.2. Ошибки ассемблирования

Для ограничений 2.3.2.2 и 2.3.2.3 пользователь может задавать специализированный текст сообщений об ошибке и критичность ошибки (warning или error). Существует возможность задавать как индивидуальные тексты для каждого ограничения, так и ссылаться на общую таблицу сообщений. Данный подход предоставляет пользователю возможность точной настройки диагностики ошибок ассемблирования с детализированными описаниями, что является необходимым условием промышленной эксплуатации кросс-системы.

## *3. Динамическая настройка кросс-системы*

В данной главе приводится краткий обзор нашей технологии динамической настройки кросс-системы для работы с моделями акселераторов, заданных в соответствии с 2.2 и 2.3 в виде *файлов описания акселераторов*.

### 3.1. Конфигурация системы

Под процессом конфигурации системы подразумевается определение конкретного набора акселераторов (типа и номера каждого акселератора). Тип акселератора (конкретная модель) определяется файлом спецификации на языке ISE. Интегрированная среда позволяет задать упорядоченный список акселераторов (файлов описания акселераторов), который и определяет конфигурацию системы, используемую для настройки соответствующих компонентов кросс-инструментария.

В рамках одной сессии интегрированной среды пользователь может много-кратно менять как конфигурацию системы, так и сами файлы описания акселе-раторов (и, конечно же, прикладную программу). Однако изменение конфигу-рации невозможно в режиме отладки, когда работает симулятор. Для смены конфигурации сессия отладки должна быть остановлена. Для редактирования файлов описания акселераторов может быть использован



визуальный front-end, который поддерживает средства анализа и верификации спецификаций (напри-мер, обнаружение конфликтующих машинных кодов для разных команд).

### 3.2. Настройка симулятора

Для настройки симулятора используется информация из файла описания акселератора, соответствующая элементам, описанным в 2.2.

В существующей реализации для настройки симулятора используется два альтернативных подхода:

- Компиляция файла описания акселератора внешним компилятором C++
- Интерпретация файла описания во время работы (run-time)

В первом случае файл описания акселератора транслируется внешним компи-лятором в динамическую библиотеку DLL. Заметим, что в этом случае синтак-сис, описанный в 2.2, трактуется как макросы, определения переменных (2.2.1) и функции (2.2.2) C++. Функция-декодер задается таблицей соответствия шаблонов машинных команд и указателей на функции команд (2.2.2.3). Инфор-мация о синтаксисе команд (2.3) не используется симулятором и для компиля-тора в этом случае представляется в виде строки инициализации специальной глобальной переменной для использования ассемблером и дисассемблером.

Программный интерфейс (API) этой библиотеки DLL содержит набор функций для извлечения информации об акселераторе. Память акселератора симулируется в виде переменных и массивов в адресном пространстве DLL. Функции поведения команд акселератора (включая операции) компилируются в исполняемый код хост-машины. Симулятор основного процессора обращается к соответствующим функциям DLL для выдачи команд акселе-ратора и инициировании очередного такта (тактовый генератор находится в отладчике). Для синхронизации потактового выполнения команд основного процессора и всех акселераторов используется модель нитей с ручным переключением контекста. Такие возможности предоставляются в операционной системе Windows в виде примитивов Fibers. Для платформы UNIX используется библиотека QuickThreads (David Keppel, 1993). Директива `FinishCycle()` в этом случае вызывает явное переключение нитей (fibers).

В случае отсутствия внешнего компилятора C++, используется другой подход, когда функции поведения команд интерпретируются внутренней виртуальной машиной во время симуляции. Однако в этом случае существует ряд ограни-чений на использование конструкций и типов языка C++ при описании модели акселератора, так как не все возможности поддерживаются интерпретатором.

## 3.3. Настройка ассемблера/дисассемблера

Для обеспечения динамической настройки ассемблера на новые команды акселераторов был разработан «универсальный» ассемблер, интерпретирующий описание синтаксиса системы команд и отображения в машинные коды, заданное в соответствии с 2.3, в процессе ассемблирования входной программы. Заметим, что часть системы команд основного процессора, не относящаяся к командам запуска акселераторов, также задается в виде 2.3. Таким образом, кроме собственно текста прикладной программы, входной информацией для ассемблера является набор файлов описания для системы команд ядра процессора и всех акселераторов системы. Объединение этих описаний задает общий ассемблерный синтаксис всех команд системы «процессор + акселераторы».

В конкретной реализации, информация о синтаксисе извлекается ассемблером либо из указанных в параметрах командной строки библиотек DLL или непосредственно из файлов описания акселераторов. При этом описание системы команд ядра основного процессора зашито в самом ассемблере, так как она не изменяется пользователем.

Заметим, что в текущей версии динамически настраиваемым является только синтаксис отдельных команд. Общий синтаксис ассемблерного файла фиксирован:

- Секции
- Объявление переменных
- Выражения
- Конструкции для макропроцессора
- Отладочная информация С-компилятора

Дисассемблер извлекает информацию о синтаксисе команд из набора файлов описания также динамически во время своей работы (in run-time).

## 3.4. Настройка отладчика

Под настройкой отладчика подразумевается настройка соответствующих окон для отображения состояния памяти акселераторов во время симуляции, включая разбиение на именованные области. С помощью отладчика пользователь также может вручную изменять значения определенных ячеек.

Во время старта отладочной сессии отладчик считывает информацию о структуре памяти акселераторов (см. 2.2.1) из соответствующих библиотек DLL или непосредственно из файлов описания акселераторов. Также для настройки подсветки синтаксиса ассемблерных команд в редакторе используется инфор-мация из синтаксической части (см. 2.3). Оттуда же берет информацию и один из профилировщиков (а именно, профилировщик покрытия системы команд).



# *4. Смежные работы*

Для спецификации аппаратуры на низком уровне используются языки HDL (Hardware Description Languages), наиболее известными из которых являются **Verilog** [6] и **VHDL** [7]. Задачей этих языков является создание спецификации, пригодной для синтеза реальной аппаратуры. Поэтому, задание аппаратуры на данном уровне абстракции является трудоемким и непригодным для быстрого проведения *исследования альтернатив дизайна*, далее DSE (Design Space Exploration). Также автоматическое построение кросс-системы затруднительно на основании описания HDL, так как информация о системе команд не определяется явно (см. [1]).

Интересный подход для моделирования аппаратуры разрабатывается в рамках программы Open SystemC Initiative [8], первоначально представленной в 2000 году. В настоящее время все активности по **SystemC** спонсируются и управляются комитетом из индустриальных компаний: ARM, Cadence, CoWare, Fujitsu, Mentor, Motorola, NEC, Sony, ST, Synopsys. SystemC это библиотека C++ классов, которая упрощает создание поведенческих моделей аппаратных систем за счет предоставления набора макросов и классов, реализующих элементы аналогичные конструкциям HDL. С помощью этих конструкций (см. [9]) возможно структурное описание системы, используя понятия модулей, процессов, портов, сигналов, интерфейсов, событий и т.п. Библиотека также предоставляет набор типов, удобных для моделирования аппаратных элементов, таких как битовые строки и числа с фиксированной точкой. Однако модель системы на SystemC предназначена только для симуляции и последующего синтеза аппаратуры, система команд явно не выделяется и построение полной кросс-системы на основании модели на SystemC затруднительно (см. [5]). В этом смысле SystemC относится скорее к классу HDL языков, при этом существуют автоматические конвертеры из Verilog и VHDL в SystemC ([10] и [11]). Скорость симуляции моделей SystemC невелика ввиду слишком низкого уровня описания деталей аппаратуры, несущественных для кросс-системы. Однако заметим, что для описания поведения отдельных операций аппаратуры примитивы SystemC очень удобны, аналогичные конструкции предоставляются и в рассматриваемой работе для описания операций и команд (см. 0). В частности предоставляются типы данных, аналогичные SystemC; также синхронизация между командами аналогична синхронизации между процессами SystemC (`FinishCycle()` аналогична функции `wait()` в SystemC).

Для решения задачи автоматической генерации компонентов кросс системы предназначены языки класса ADL (Architecture Description Languages). Дополнительная информация и полный обзор языков ADL может быть найден в [1] - [5]. Здесь приведем только наиболее известные решения.

Одним из первых языков ADL был **nML** [12], изначально разработанный в Техническом Университете Берлина, Германия (1991). nML использовался в качестве способа описания аппаратуры для симулятора SIGH/SIM и компилятора CBC (с языка ALDiSP). В nML система команд процессора

описывается с помощью атрибутных грамматик. Атрибуты включают в себя поведение (action), ассемблерный синтаксис (syntax) и отображение в машинные коды (image). Оригинальный nML не содержит механизмов описания многотактовых команд. Однако nML получил дальнейшее развитие в бельгийском научно-исследовательском центре микроэлектроники IMEC, где в рамках дочерней компании Target Compiler Technologies была создана коммерческая среда разработки [13]-[14], ориентированная на DSP архитектуры (1995). В эту среду входят компилятор CHESS (с языка C), симулятор CHECKERS, ассемблер, дисассемблер и линкер. Также поддерживается синтез VHDL описания. В рамках этой коммерческой среды компания Target Compiler Technologies модифицировала nML для поддержки более сложной аппаратуры (в частности введены механизмы явного описания конвейера), хотя из маркетинговых заявлений компании (технические спецификации недоступны) до конца не ясно, какие именно средства описания ILP поддерживаются. Также nML поддерживает только команды фиксированной длительности и производительность симулятора, опубликованная в [14], невысока.

Последователем nML стал язык **Sim-nML** [15], работы над которым ведутся с 1998 года в Индийском Технологическом Институте (Indian Institute of Technology Kanpur) при поддержке компании Cadence. Главным принципиальным нововведением стал дополнительный атрибут использования ресурсов (uses) в грамматике описания команд. Это позволяет описывать использование ресурсов и, тем самым, обнаруживать конфликты между командами. В рамках проекта Sim-nML были разработаны кодогенератор для компилятора, симулятор, ассемблер и дисассемблер. К сожалению, отсутствует интегрированная среда разработки и отладки.

Язык **ISDL** был разработан в университете MIT, США [16] и представлен на конференции по автоматизированному дизайну DAC [17] в 1997 году. Основной специализацией ISDL является описание VLIW архитектур. Изначально задумывалась реализация компилятора, ассемблера и симулятора, а также генератора Verilog описания. Аналогично nML, ISDL главным образом описывает систему команд процессора, включающую в себя семантику поведения, синтаксис ассемблера, машинные коды, а также описание ресурсных конфликтов, используя атрибутную грамматику. К важным достоинствам языка можно отнести возможность точно специфицировать задержки и конфликтные ситуации для ILP в виде логических правил, хотя явное описание конвейера отсутствует. Несмотря на довольно продуманный язык, к сожалению, отсутствуют в доступном виде реальные утилиты, поддерживающие его. Инициаторы проекта ограничились только реализацией ассемблера и некоторых модулей симулятора и кодогенератора для компилятора в качестве диссертационных работ MIT.

Язык **EXPRESSION** [18]-[19] разработан в Университете Калифорнии (University of California, Irvine, США), впервые был представлен на конференции DATE в 1999 году. Этот язык поддерживает широкий класс встроенных систем с ILP и иерархиями памяти от RISC, DSP, ASIP до VLIW.



EXPRESSION содержит интегрированное описание структуры и поведения подсистемы процессор-память. Спецификация на EXPRESSION состоит из шести секций (первые три отвечают за поведение, последние три за структуру):

- Спецификация операций (набор атомарных команд с кодами, описанием параметров и семантики (поведения)).
- Описание формата команды (команда состоит из ячеек, ответственных за определенный функциональный модуль, которые могут заполняться атомарными операциями для параллельного выполнения).
- Отображение общих операций компилятора на машинные операции, описанные в первой секции. Данное описание используется для кодогенератора компилятора.
- Описание компонент (функциональные устройства, шины, порты и т.п.).
- Описание конвейера и связей компонентов.
- Описание иерархии памяти (регистровая память, кэш, SRAM, DRAM).

Из описания EXPRESSION автоматически генерируются компилятор EXPRESS и симулятор SYMPRESS. К недостаткам решения на основе EXPRESSION следует отнести невысокую скорость симуляции и относительную трудоемкость описания (из-за наличия детальной структурной составляющей). В этом смысле EXPRESSION стоит между чистыми поведенческими ADL решениями (типа nML) и структурными описаниями HDL уровня.

Язык **LISA** [21]-[22] разрабатывался в университете RWTH Aachen (Германия) изначально в качестве средства описания аппаратуры для генерации симуляторов. Первые результаты работ по проекту LISA были опубликованы в 1996 году. Первоначальной целевой архитектурой были DSP процессоры. К ключевым характеристикам LISA можно отнести подробное описание конвейера на уровне операций с возможностью задания зависимостей и блокировок. Конвейерные конфликты задаются явно. Каждая команда задается в виде набора операций, которые определяются как регистровые пересылки за время одного кванта синхронизации. Описание LISA состоит из двух основных частей: спецификации ресурсов и описания операций. Описание операций в свою очередь содержит следующие секции:

- DECLARE (определение объектов и групп через другие объекты и операции - фактически правила грамматики).
- CODING (описание бинарного кодирования операции).
- SYNTAX (описание ассемблерного синтаксиса и параметров).
- BEHAVIOR и EXPRESSION (описание поведения операции в виде кода на языке C/C++).
- ACTIVATION (описание задержек (timings) и поведения конвейера).

К сожалению, отсутствует полная публичная спецификация языка LISA, вся информация взята из различных статей.

Согласно [20], из всех представленных подходов только система на основе языка EXPRESSION предоставляет средства для описания акселераторов. Однако, она ориентирована только на проведение фазы DSE, в ней отсутствуют такие производственные компоненты как ассемблер, дисассемблер, отладчик. Кроме того, ни одна из систем не поддерживает динамическую настройку компонентов, так как все компоненты создаются специальными генераторами кода в виде исходного кода на C/C++ и необходимо использование внешних компиляторов для получения готовой кросс системы. Настройка обработки ошибок в ассемблере также не поддерживается указанными языками.

## *5. Заключение*

В данной статье представлена технология динамической настройки кросс-системы для поддержки ассемблирования, симуляции и отладки программ, содержащих команды, неизвестные на этапе построения основного инструмен-тария. Семантика и синтаксис этих команд определяются конкретными акселе-раторами, создаваемыми пользователями при построении специфической конфигурации системы «процессор + акселераторы» и неизвестными производителю основного процессора (и соответственно кросс-системы).

Для решения этой задачи была разработана абстрактная модель поддерживаемых акселераторов и интерфейса с процессором, охватывающая широкий спектр возможной аппаратуры. Предложенный язык спецификации ISE позволяет пользователям описывать спецификации конкретных акселераторов в рамках этой абстрактной модели. Созданные файлы спецификаций регистрируются в настройках интегрированной среды при описании конфигурации системы (см. 3.1). При этом компоненты кросс-инструментария настраиваются в соответствии с этими спецификациями динамически (in-run-time). В результате прикладные программисты получают возможность писать и отлаживать программы с использованием новых команд. Спецификации и конфигурация акселераторов в системе могут многократно меняться в рамках одного сеанса интегрированной среды, в том числе с помощью визуальных средств редактирования, анализа и верификации.

На основе описанной технологии в рамках коммерческого проекта в ИСП РАН была реализована настраиваемая кросс-система для DSP процессора заказчика (поддерживающего акселераторы). Авторам известно, что с помощью этой сис-темы пользователями были созданы рабочие модели реальных акселераторов:

a. Быстрого преобразования Фурье
b. Алгоритмов эхо подавления
c. Операций с комплексными числами



d. Операций кодирования видео
e. Операций цифровой фильтрации звука

Пиковая производительность симулятора центрального процессора (на хост-машине PIII-1000MHz) в рассматриваемой системе составляет порядка 10 миллионов тактов в секунду. При использовании конфигурации с одним акселератором (эхо подавления) производительность составила порядка 1 миллиона тактов в секунду (при этом симулируются процессор и акселератор, работающие параллельно), что обусловлено большими потерями на синхро-низацию процессов выполнения процессора и акселератора.

Дальнейшие наши работы в этой области направлены на расширение языка ISE и соответствующих утилит для поддержки моделирования полной системы, включая описание центрального процессора. Отдельное внимание уделяется увеличению производительности симулятора за счет использования JIT технологий и использования знаний о конкретной программе. Также предполагается расширить возможности системы для настройки компилятора с языка высокого уровня для генерации кода с учетом наличия акселераторов (в настоящее время команды акселератора на уровне языка C используются вручную в виде ассемблерных вставок).

**Литература**


[1] Hiroyuki Tomiyama, Ashok Halambi, Peter Grun. Architecture Description Languages for Systems-on-Chip Design. Center for Embedded Computer Systems, Univertsity of California. 2000.
[2] Wei Qin, Sharad Malik. Architecture Description Languages for Retargetable Compilation. The Compiler Design Handbook, CRC Press, 2003.
[3] Clifford Liem, Pierre G. Paulin, Ahmed A.Jerraya. Retargetable Compilers for Embedded Core Processors. Kluwer Academic Publishers, 1997.
[4] Rainer Leupers. Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997.
[5] Lin Yung-Chia. Hardware/Software Co-design with Architecture Description Language. Programming Language Lab. NTHU. 2003.
[6] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Std 1364-1995.
[7] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987
[8] Open SystemC Initiative. http://www.systemc.org/
[9] SystemC User's Guide. Version 2.0.1. http://www.systemc.org/
[10] N.Agliada, A.Fin, F.Fummi, M.Martignano, G.Pravadelli. On the Reuse of VHDL Modules into SystemC Designs, FDL 2001
[11] Leila Mahmoudi Ayough Ali Haj Abutalebi Omid F. Nadjarbashi Shaahin Hessabi. Verilog2SC: A Methodology for Converting Verilog® HDL to SystemC. HDLCon 2002
[12] A. Fauth, J. Van Praet, M. Freericks. Describing Instruction Set Processors Using nML. Proc European Design and Test Conf., Paris, March 1995.
[13] Chess/Checkers Products. Target Compiler Technology. http://www.retarget.com/
[14] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. Alta Group of Cadence Design Systems, Inc. DAC 1997.

[15] Sim-nML Homepage. http://www.cse.iitk.ac.in/sim-nml/
[16] ISDL Project Homepage. http://caa.lcs.mit.edu/caa/home.html
[17] George Hadjiyannis, Silvina Hanono. ISDL: An Instruction Set Description Language for Retargetability. Srinivas Devadas. Department of EECS, MIT. DAC 1997.
[18] EXPRESSION Homepage. http://www.cecs.uci.edu/~aces/index.html
[19] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt and Alex Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability, DATE 99.
[20] Prabhat Mishra, Frederic Rousseau, Nikil Dutt, Alex Nicolau. Architecture Description Language Driven Design Space Exploration in the Presence of Coprocessors. SASIMI 2001.
[21] V. Zivojnovic, S. Pees, and H. Meyr. LISA: Machine description language and generic machine model for HW/SW co-design. In Proc. of Workshop on VLSI Signal Processing, 1996.
[22] Andreas Hoffmann, Achim Nohl, Stefan Pees, Gunnar Braun, Heinrich Meyr. Generating Production Quality Software Development Tools Using a Machine Description Language. DATE 2001.
[23] *N. Cutland*. Computability: an introduction to recursive function theory. Cambridge University Press, 1980.